AsyncTask

AsyncTask разработан, чтобы быть вспомогательным классом вокруг Tread и Handler

Содержание

- Асинхронная задача
- Использование
- Универсальные типы AsyncTask
- 4 этапа асинхронной задачи
- Threading rules
- Наблюдаемость памяти
- Версии сборки и Порядок выполнения
- Пример из Климова
- Обзор полей и методов класса AsyncTask

AsyncTask

AsyncTask позволяет правильно и легко использовать поток пользовательского интерфейса. Этот класс позволяет выполнять фоновые операции и публиковать результаты в потоке пользовательского интерфейса и манипулировать потоками и/или обработчиками.

AsyncTask разработан, чтобы быть вспомогательным классом вокруг Tread и Handler и не представляет собой универсальную структуру потоков.

AsyncTasks в идеале должны использоваться для **коротких операций** (не более нескольких секунд.)

Если вам нужно поддерживать потоки в течение длительного периода времени, настоятельно рекомендуется использовать различные API, предоставляемые **java.util.concurrent** пакетом, такие как

Executor,

ThreadPoolExecutor и

FutureTask.

Асинхронная задача

Асинхронная задача определяется вычислением, которое выполняется в фоновом потоке и результат которого публикуется в потоке пользовательского интерфейса.

Асинхронная задача определяется 3 универсальными типами, называемыми

- Params,
- Progress и
- Result,
- и 4 шагами, называемыми
- onPreExecute,
- doInBackground,
- onProgressUpdate и
- onPostExecute.

Использование

AsyncTask должен быть разделен на подклассы, чтобы быть использованным.

Подкласс переопределит по крайней мере один метод (doInBackground(Params...)), и чаще всего будет переопределять второй (onPostExecute (Result).)

Пример подклассов:

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
        protected Long doInBackground(URL... urls) {
            int count = urls.length;
            long totalSize = 0;
            for (int i = 0; i < count; i++) {
                totalSize += Downloader.downloadFile(urls[i]);
                publishProgress((int) ((i / (float) count) * 100));
                // Escape early if cancel() is called
                if (isCancelled()) break;
            }
            return totalSize;
        }
 protected void onProgressUpdate(Integer... progress) {
            setProgressPercent(progress[0]);
        }

        protected void onPostExecute(Long result) {
            showDialog("Downloaded " + result + " bytes");
        }
    }
```

Универсальные типы Asynctask

Асинхронная задача использует следующие три типа:

- Params-тип параметров, передаваемых задаче при выполнении.
- Progress, тип единиц прогресса, опубликованных во время фонового вычисления.
- Result, тип результата фонового вычисления.

Не все типы всегда используются асинхронной задачи.

Чтобы пометить тип как неиспользуемый, просто используйте тип Void:

```
private class MyTask extends AsyncTask<Void, Void, Void> { ... }
```

Работа с **AsyncTask**

Напрямую с классом **AsyncTask** работать нельзя (абстрактный класс), вам нужно наследоваться от него (extends).

Ваша реализация должна предусматривать классы для объектов, которые будут переданы в качестве параметров методу **execute()**, для переменных, что станут использоваться для оповещения о ходе выполнения, а также для переменных, где будет храниться результат.

Формат такой записи следующий:

AsyncTask<[Input_Parameter Type], [Progress_Report Type], [Result Type]>

Если не нужно принимать параметры, обновлять информацию о ходе выполнения или выводить конечный результат, просто укажите тип **Void** во всех трёх случаях.

В параметрах можно использовать только обобщённые типы (Generic), т.е. вместо **int** используйте **Integer** и т.п.

Соответственно, варианты могут быть самыми разными:

AsyncTask<Void, Void, Void>

AsyncTask<String, Integer, Integer>

AsyncTask<String, Void, Integer>

AsyncTask<String, Integer, String>

Для запоминания можно смотреть на схему.

```
private class DownloadImage extends AsyncTask<String, Integer, Bitmap> {

    @Override
    protected void onPreExecute() {...}

    @Override
    protected Bitmap doInBackground(String... params) {...}

    @Override
    protected void onProgressUpdate(Integer... values) {
        super.onProgressUpdate(values);
    }

    @Override
    protected void onPostExecute(Bitmap bitmap) {...}
```

Каркас операции

```
private class MyAsyncTask extends AsyncTask<String, Integer, Integer> {
@Override protected Integer doInBackground(String... parameter)
{ int myProgress = 0;
 // [... Выполните задачу в фоновом режиме, обновите переменную
    myProgress...]
publishProgress(myProgress);
 // [... Продолжение выполнения фоновой задачи ...]
// Верните значение, ранее переданное в метод onPostExecute
return result; }
 @Override protected void onProgressUpdate(Integer... progress) {
// [... Обновите индикатор хода выполнения, уведомления или другой
// элемент пользовательского интерфейса ...]
 }
@Override protected void onPostExecute(Integer... result) {
 // [... Сообщите о результате через обновление пользовательского //
    интерфейса, диалоговое окно или уведомление ...]
 } }
```

4 шага

При выполнении асинхронной задачи она проходит 4 этапа:

- **onPreExecute(),** вызывается в потоке пользовательского интерфейса перед выполнением задачи. Этот шаг обычно используется для настройки задачи, например, путем отображения индикатора выполнения в пользовательском интерфейсе.
- **doInBackground(params ...),** вызывается в фоновом потоке сразу после завершения выполнения функции **onPreExecute ().**

Этот шаг используется для выполнения фоновых вычислений, которые могут занять много времени.

Параметры асинхронной задачи передаются на этот шаг.

Результат вычисления должен быть возвращен на этом шаге и будет передан обратно на последний шаг.

На этом шаге также можно использовать **publishProgress (Progress...)** опубликовать одну или несколько единиц прогресса.

Эти значения публикуются в потоке пользовательского интерфейса в **onProgressUpdate (Progress...)** шаге.

4 шага

- **onProgressUpdate (Progress ...),** вызывается в потоке пользовательского интерфейса после вызова метода **publishProgress (Progress...).**

Время выполнения не определено.

Этот метод используется для отображения любой формы прогресса в пользовательском интерфейсе, в то время как фоновое вычисление все еще выполняется.

Например, его можно использовать для анимации индикатора выполнения или отображения журналов в текстовом поле.

- **onPostExecute (Result),** вызывается в потоке пользовательского интерфейса после завершения фонового вычисления. Результат фонового вычисления передается на этот шаг в качестве параметра.

Отмена задачи

Задачу можно отменить в любое время, вызвав команду cancel (boolean).

Вызов этого метода приведет к тому, что последующие вызовы функции is Cancelled () вернут true.

После вызова этого метода, onCancelled(java.lang.Object), вместо onPostExecute (java.lang.Object) будет вызван после doInBackground(java.lang.Object[])

Чтобы гарантировать, что задача отменяется как можно быстрее, вы всегда должны периодически проверять возвращаемое значение isCancelled () из doInBackground(java.lang.Object[]), если это возможно (например, внутри цикла.)

Threading rules

Существует несколько правил потоковой передачи, которые необходимо соблюдать для правильной работы этого класса:

- Класс AsyncTask должен быть загружен в поток пользовательского интерфейса. Это делается автоматически с момента сборки.**VERSION_CODES.JELLY_BEAN**.
- Экземпляр задачи должен быть создан в потоке пользовательского интерфейса.
- **execute(Params...)** должен быть вызван в потоке пользовательского интерфейса.
- **Не вызывайте onPreExecute (), onPostExecute(Result), doInBackground(Params...), onProgressUpdate(Progress...)** вручную.
- Задача может быть выполнена только один раз (исключение будет выдано при попытке второго выполнения.)

Наблюдаемость памяти

AsyncTask гарантирует, что все обратные вызовы синхронизируются, чтобы гарантировать следующее без явных синхронизаций.

- Эффекты памяти onPreExecute () и все остальное, выполненное до вызова execute(Params...), включая конструкцию объекта AsyncTask, видны doInBackground (Params...).
- Эффекты памяти doInBackground (Params...) видны onPostExecute (результат).
- Любые эффекты памяти doInBackground (Params...) предшествующий вызову publishProgress(прогресс...) видны соответствующему вызову onProgressUpdate (Progress..) .
- (Но doInBackground (Params...) продолжает работать, и необходимо позаботиться о том, чтобы более поздние обновления в doInBackground(Params...) не вмешивайтесь в процесс вызова onProgressUpdate (Progress..))
- Любые эффекты памяти, предшествующие вызову cancel (boolean), видны после вызова is Cancelled (), который возвращает true в результате, или во время и после результирующего вызова onCancelled ().

Порядок выполнения

При первом введении **AsyncTasks** выполнялись последовательно в одном фоновом потоке.

Начиная с Build.VERSION_CODES.DONUT , это было изменено на пул потоков, позволяющих нескольким задачам работать параллельно.

Начиная с сборки.VERSION_CODES.HONEYCOMB, задачи выполняются в одном потоке, чтобы избежать распространенных ошибок приложений, вызванных параллельным выполнением.

Если вы действительно хотите параллельного выполнения, вы можете вызвать

executeOnExecutor(java.util.concurrent.Executor, java.lang.Object[]) со THREAD_POOL_EXECUTOR.

Кот полез на крышу (А. Климов)

Напишем простой пример с использованием названных методов. Предположим, мы хотим описать процесс восхождения котов на крышу.

Выведем на экран слово **Полез на крышу** в методе **onPreExecute()**, эмулируем тяжёлый код в методе **doInBackground()**, выведем на экран слово **Залез** в методе **onPostExecute()**.

Создадим новый проект и добавим на экран кнопку, индикатор прогресса и текстовую метку:

- <Button android:id="@+id/button_start" android:layout_width="wrap_content" android:layout_height="wrap_content" android:onClick="onClick" android:text="Поехали" />
- <ProgressBar android:id="@+id/progressbar" android:layout_width="wrap_content" android:layout_height="wrap_content" />
- <TextView android:id="@+id/text_info" android:layout_width="wrap_content" android:layout_height="wrap_content" android:text="" />

При щелчке кнопки должна запуститься тяжёлая задача.

Это может быть загрузка файла из сети, обработка изображения, сохранение больших данных в файл или в базу данных.

В нашем случае - кот полез на крышу.

В **TextView** будем выводить текущую информацию.

Компонент **ProgressBar** будет показывать, что приложение не зависло во время выполнения задачи.

class MainActivity

```
import android.os.AsyncTask;
 import android.os.Bundle;
 import android.view.View;
 import android.widget.TextView;
import androidx.appcompat.app.AppCompatActivity;
 import java.util.concurrent.TimeUnit;
 public class MainActivity extends AppCompatActivity {
private TextView mInfoTextView;
@Override
protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main);
 mInfoTextView = findViewById(R.id.text_info); }
 public void onClick(View view) {
 CatTask catTask = new CatTask();
catTask.execute(); }
```

class CatTask

```
class CatTask extends AsyncTask<Void, Void, Void> {
@Override
 protected void onPreExecute() { super.onPreExecute();
mInfoTextView.setText("Кот полез на крышу");
startButton.setVisibility(View.INVISIBLE); // прячем кнопку
}
@Override
 protected Void doInBackground(Void... voids) {
 try{ TimeUnit.SECONDS.sleep(5); }
catch (InterruptedException e){ e.printStackTrace(); }
return null; }
@Override
 protected void onPostExecute(Void aVoid) {
super.onPostExecute(aVoid);
mInfoTextView.setText("Кот залез на крышу");
startButton.setVisibility(View.VISIBLE); // показываем кнопку
 } } }
```

Комментарий

- Запустите проект и нажмите на кнопку. Сначала появится текст "Кот полез на крышу", который через 5 секунд сменится надписью "Кот залез на крышу". Индикаторе прогресса при этом будет постоянно крутиться.
- У кота одна задача - залезть на крышу. Поэтому мы создали новую задачу **CatTask**, которая наследуется от **AsyncTask**. Для первого примера мы использовали **Void**, чтобы пока не связываться с параметрами.
- В методе **onPreExecute()** мы устанавливаем начальный текст перед выполнением задачи.
- В методе **doInBackground()** идёт имитация тяжёлой работы. Здесь нельзя писать код, связанный с пользовательским интерфейсом
- В методе **onPostExecute()** мы выводим сообщение, которое появится после выполнения задачи.

- Я небо помыл !
Ща солнышко выйдет !!!

Summary

- Nested classes
- Enum AsyncTask.Status Состояние - текущее состояние задачи
- Fields
- public static finalExecutorSERIAL_EXECUTOR An Executor , который выполняет задачи в последовательном порядке.
- public static finalExecutorTHREAD_POOL_EXECUTOR An Executor исполнитель который выполняет задачи в параллельном порядке..
- Public constructors
- AsyncTask() Creates a new asynchronous task.

Public methods

final boolean	cancel(boolean mayInterruptIfRunning) Attempts to cancel execution of this task.
finalAsyncTask<Params, Progress, Result>	execute(Params... params)Executes the task with the specified parameters.
static void	execute(Runnable runnable)Convenience version of execute(java.lang.Object) for use with a simple Runnable object.
finalAsyncTask<Params, Progress, Result>	executeOnExecutor(Executor exec, Params... params)Executes the task with the specified parameters.
final Result	get(long timeout, TimeUnit unit)Waits if necessary for at most the given time for the computation to complete, and then retrieves its result.
final Result	get()Waits if necessary for the computation to complete, and then retrieves its result.
final AsyncTask.Status	getStatus()Returns the current status of this task.
final boolean	isCancelled()Returns true if this task was

Protected methods

abstract Result	doInBackground(Params... params)Override this method to perform a computation on a background thread.
void	onCancelled() Applications should preferably override onCancelled(java.lang.Object).
void	onCancelled(Result result) Runs on the UI thread after cancel(boolean) is invoked anddoInBackground(java.lang.Object[]) has finished.
void	onPostExecute(Result result) Runs on the UI thread after doInBackground(Params...).
void	onPreExecute()Runs on the UI thread before doInBackground(Params...).
void	onProgressUpdate(Progress... values)Runs on the UI thread after publishProgress(Progress...) is invoked.
final void	publishProgress(Progress... values)This method can be invoked from doInBackground(Params...) to publish updates on the UI thread while the background computation is still running.

Fields&constructor

SERIAL_EXECUTOR
Added in API level 11
public static final Executor SERIAL_EXECUTOR
Исполнитель, выполняющий задачи по очереди в последовательном порядке. Эта сериализация является глобальной для конкретного процесса..

THREAD_POOL_EXECUTOR
Added in API level 11
public static final Executor THREAD_POOL_EXECUTOR
Исполнитель, который может использоваться для параллельного выполнения задач.

Public constructors
AsyncTask
Added in API level 3
public AsyncTask ()
Создает новую асинхронную задачу. Этот конструктор должен быть вызван в потоке пользовательского интерфейса.

public final boolean cancel (boolean mayInterruptIfRunning)

Пытается отменить выполнение этой задачи. Эта попытка будет неудачной, если задача уже выполнена, уже отменена или не может быть отменена по какой-либо другой причине.

В случае успеха и если эта задача не была запущена при вызове функции отмена, она никогда не должна выполняться.

Если задача уже запущена, то параметр mayInterruptIfRunning определяет, должен ли поток, выполняющий эту задачу, быть прерван при попытке остановить задачу.

Вызов этого метода приведет к **onCancelled(java.lang.Object)** вызывается в потоке пользовательского интерфейса после **doInBackground (java.lang.Object[]).**

Вызов этого метода гарантирует, что onPostExecute(Object) никогда впоследствии не вызывается, даже если cancel возвращает false, но onPostExecute(Result) еще не запущен.

Чтобы завершить задачу как можно раньше, периодически проверяйте isCancelled () из doInBackground(java.lang.Object[]).

Это только требует отмены. Он никогда не ожидает завершения выполнения фоновой задачи, даже если mayInterruptIfRunning имеет значение true.

public final AsyncTask<Params, Progress, Result> **execute** (Params... params)

Примечание: эта функция планирует задачу в очереди для одного фонового потока или пула потоков в зависимости от версии платформы.

При первом введении AsyncTasks выполнялись последовательно в одном фоновом потоке. Начиная с сборки.VERSION_CODES.Donut, это было изменено на пул потоков, позволяющих нескольким задачам работать параллельно.

Начало Сборки.VERSION_CODES.HONEYCOMB, задачи возвращаются к выполнению в одном потоке, чтобы избежать распространенных ошибок приложений, вызванных параллельным выполнением.

Если вы действительно хотите параллельного выполнения, вы можете использовать executeOnExecutor(Executor, Params...) версия этого метода с THREAD_POOL_EXECUTOR; однако, смотрите комментарий там для предупреждений о его использовании.

Пример

```
@Override public void onClick(View view) {
    asyncTask1 = new MyAsyncTask(mProgressBar1);
    asyncTask1.execute();
 asyncTask2 = new MyAsyncTask(mProgressBar2);
    asyncTask2.execute();
asyncTask3 = new MyAsyncTask(mProgressBar3);
    asyncTask3.execute();
 asyncTask4 = new MyAsyncTask(mProgressBar4);
    startAsyncTaskInParallel(asyncTask4);
 asyncTask5 = new MyAsyncTask(mProgressBar5);
    startAsyncTaskInParallel(asyncTask5); } }); } }
```

public static void execute (Runnable runnable)

Удобен в варианте execute(java.lang.Object) для использования с простым запускаемым объектом.

Этот метод должен быть вызван из Looper#getMainLooper () вашего приложения.

public final class Looper extends Object

↳android.os.Looper

Класс, используемый для запуска цикла сообщений для потока.

Потоки по умолчанию не имеют связанного с ними цикла сообщений;

чтобы создать его, вызовите prepare() в потоке, который должен запустить цикл, а затем loop (), чтобы он обрабатывал сообщения до тех пор, пока цикл не будет остановлен.

Большинство взаимодействий с циклом сообщений осуществляется через класс обработчика.

Это типичный пример реализации потока петлителя, использующего разделение prepare() и loop() для создания начального обработчика для связи с looper.

```
class LooperThread extends Thread {
public Handler mHandler;
 public void run() { Looper.prepare();
 mHandler = new Handler() {
public void handleMessage(Message msg) {
 // process incoming
 messages here } };
Looper.loop(); } }
```

public final AsyncTask<Params, Progress, Result> executeOnExecutor (Executor exec, Params... params)

Выполняет задачу с указанными параметрами. Задача возвращает себя (this), так что вызывающий может сохранить ссылку на него.

Этот метод обычно используется с THREAD_POOL_EXECUTOR, чтобы разрешить параллельное выполнение нескольких задач в пуле потоков, управляемых AsyncTask, однако вы также можете использовать свой собственный исполнитель для пользовательского поведения.

Предупреждение: разрешение параллельного выполнения нескольких задач из пула потоков-это, как правило, не то, что нужно, потому что порядок их работы не определен.

Например, если эти задачи используются для изменения любого общего состояния (например, записи файла из-за нажатия кнопки), нет никаких гарантий относительно порядка изменений.

Без тщательной работы в редких случаях более новая версия данных может быть перезаписана более старой, что приводит к неясным потерям данных и проблемам стабильности.

Пример

```
@TargetApi(Build.VERSION_CODES.HONEYCOMB)
private void startAsyncTaskInParallel(MyAsyncTask task) {
 if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB)
   task.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR);
else task.execute(); }
```

protected abstract Result doInBackground (Params... Params)

Переопределите этот метод для выполнения вычисления в фоновом потоке.

Указанными параметрами являются параметры, передаваемые для выполнения в execute(Params...) вызывающим эту задачу.

Обычно он выполняется в фоновом потоке.

Но чтобы лучше поддерживать платформы тестирования, рекомендуется, чтобы он также допускал прямое выполнение на переднем плане потока, как часть execute (Params...).

Этот метод может вызвать publishProgress (Progress...) для публикации обновлений в потоке пользовательского интерфейса.

Выполнение этого метода может занять несколько секунд, поэтому его следует вызывать только из рабочего потока.

Пример

```
public class MyAsyncTask extends AsyncTask<Void, Integer, Void> {
 private ProgressBar mProgressBar;
 public MyAsyncTask(ProgressBar target) {
mProgressBar = target; }
 @Override
protected Void doInBackground(Void... params) {
 for (int i = 0; i < 100; i++) {
 publishProgress(i);
SystemClock.sleep(100); }
 return null; }
@Override
 protected void onProgressUpdate(Integer... values) {
    mProgressBar.setProgress(values[0]); } } }
```

protected void onPostExecute (Result result)

Выполняется в потоке пользовательского интерфейса после doInBackground(Params...).

Указанный результат - это значение, возвращаемое параметром doInBackground(Params...).

Чтобы лучше поддерживать платформы тестирования, рекомендуется, чтобы это было написано, чтобы разрешить прямое выполнение как часть вызова execute ().

Версия по умолчанию ничего не делает.

Этот метод не будет вызван, если задача была отменена.

protected void onPreExecute ()

Выполняется в потоке пользовательского интерфейса перед doInBackground(Params...).

Вызывается непосредственно execute (Params...) или executeOnExecutor(Executor, Params...). Версия по умолчанию ничего не делает.

Этот метод должен быть вызван из Looper#getMainLooper () вашего приложения.

protected void onProgressUpdate (Progress... values)

- Выполняется в потоке пользовательского интерфейса после вызова publishProgress(Progress...).
- Указанные значения являются значениями, переданными в publishProgress (Progress...). Версия по умолчанию ничего не делает.
- Этот метод должен быть вызван из Looper#getMainLooper () вашего приложения.

protected final void publishProgress (Progress... values)

Этот метод может быть вызван из doInBackground (Params...) для публикации обновлений в потоке пользовательского интерфейса во время выполнения фоновых вычислений.

Каждый вызов этого метода инициирует выполнение onProgressUpdate (Progress...) в потоке пользовательского интерфейса. onProgressUpdate (прогресс...) не будет вызван, если задача была отменена.

Выполнение этого метода может занять несколько секунд, поэтому его следует вызывать только из рабочего потока

Литература

- https://developer.android.com/reference/android/os/AsyncTask.html
- http://developer.alexanderklimov.ru/android/theory/asynctask.php
- https://android-tools.ru/coding/asynctask-ustarel-chto-teper/